Cloud Computing - Ex2

A discussion on system scaling- messaging system

by Ron Azuelos (207059114) and Tal Kapitolnik (308404306)

Our suggested Architecture

In our implementation of the messaging system backed service we took into consideration the scaling factor of the system, and therefore we chose AWS services that are serverless ang support autoscaling- API Gateway, dynamoDB and Lambda. Using those services , the deployment can support both low and high scales, and the cost will be in correlation to the scale.
The API Gateway receives the HTTP requests from the client (the non-implemented frontend) and triggers a Lambda function execution.
The Lambda function implements and performs all the required actions for the messaging system to operate. It is implemented with Python 3.12 and includes 5 modules:

- **messaging_system.py** - the main lambda handler that is responsible to route the requests to the relevant functions.
- **user_api.py** - handles the actions that are relevant to a user- registration of new user and blocking a user by another user.
- **group_api.py** - handles the actions that are relevant to a group- creating a new group, adding and removing users from a group.
- **message_api.py** - handles the messages part- sending messages to a user or group, getting unread messages.
- **utils.py** - contains common functions of validating the client input, generating objects ids etc.

We used DynamoDB as the persistence storage with 2 tables, “users” that hold the information about the users, and “chats” that contains both groups chats and regular 2-users conversations.

Scaling per service

API Gateway

AWS API Gateways can handle up to 10,000 RPS. In a case that we need more than that, the quotas can be increased upon request to the AWS team.
In terms of pricing, we only pay for what we use, on a linear scale.

Lambda

As in the case on API gateway, there are limitations for the deployment. AWS Lambda has a default concurrency limit of 1000 concurrent executions per account per region. This quota may be enough in the case of 1000s of users in average load, but in the case of 10Ks users or more, we will be required to request a quota increase from AWS. In the case where the limit cannot be increased, we should consider using caching and other optimizations to reduce the request load, or even consider hosting the backend service in a container-based deployment (like ECS or AKS).

In terms of cost, Lambda pricing is based on the number of invocations and the duration of each function execution. With more users involved, the frequency of invocations and associated costs will increase proportionally. To optimize costs, we can optimize the code to run faster and therefore minimize execution time, or consider moving to container-based deployment.

DynamoDB

DynamoDB is designed to handle requests with single-digit millisecond performance at any scale. The default limits include 40K read capacity units and 40K write capacity units per table. In the case of 1000s of users or even the 10Ks of users, the limitations will probably be sufficient and can be managed automatically with auto-scaling and on-demand features. In the case where the current limitations will not suffice, we can use other features suggested by the DynamoDB service, like reserved capacity, which can add 1M (or more, on demand) read and write capacity units.
We can also use DynamoDB Accelerator (DAX), which is Dynamo DB's in memory cache to enhance our performance.
In terms of cost, DynamoDB pricing will scale with the rise in read/write capacity units. Using on-demand pricing may offer cost benefits for unexpected bursts in workloads. Also, there will be additional costs from using advanced features such as reserved capacity or global tables, and of course from using external caching service.